CA-BIST for Asynchronous Circuits: A Case Study on the RAPPID Asynchronous Instruction Length Decoder*

Marly Roncken[1], Ken Stevens[1], Rajesh Pendurkar[2], Shai Rotem[1], and Parimal Pal Chaudhuri[3]

[1]Intel Corporation, Hillsboro, Oregon, USA / Haifa, Israel

[2]Sun Microsystems, Palo Alto, California, USA

[3]Bengal Engineering College, Sibpur, West Bengal, India

Abstract

This paper presents a case study in low-cost non-invasive Built-In Self Test (BIST) for RAPPID, a large-scale 120,000-transistor asynchronous version of the Pentium® Pro Instruction Length Decoder, which runs at 3.6 GHz. RAPPID uses a synchronous 0.25 micron CMOS library for static and domino logic, and has no Design-for-Test hooks other than some debug features. We explore the use of Cellular Automata (CA) for on-chip test pattern generation and response evaluation. More specifically, we look for fast ways to tune the CA-BIST to the RAPPID design, rather than using pseudo-random testing. The metric for tuning the CA-BIST pattern generation is based on an abstract hardware description model of the instruction length decoder, which is independent of implementation details, and hence also independent of the asynchronous circuit style. Our CA-BIST solution uses a novel bootstrap procedure for generating the test patterns, which give complete coverage for this metric, and cover 94% of the testable stuck-at faults for the actual design at switch level. Analysis of the undetected and untestable faults shows that the same fault effects can be expected for a similar clocked circuit. This is encouraging evidence that testability is no excuse to avoid asynchronous design techniques in addition to high-performance synchronous solutions.

Keywords asynchronous circuits, BIST, Cellular Automata, dynamic circuits, pulse logic, domino logic, self-timed circuits, stuck-at faults, switch-level fault simulation, testability.

*During this case study, Parimal Pal Chaudhuri was Visiting Professor with Intel in Hillsboro, and Rajesh Pendurkar — at that time with Georgia Institute of Technology, Atlanta, Georgia — worked as a Summer Student on the implementation of the CA-BIST architecture and algorithm.

1 Introduction

Built-In Self Test (BIST) as on-chip test method for random logic designs is not new [3], neither is its application to asynchronous design, pioneered by [11] and more recently investigated by [2]. Our BIST study differs from previous work in the following ways:

- **BIST state machine model**
  We use *Cellular Automata (CA)* [6] for on-chip test pattern generation and response evaluation because of the wider variety of state transition behaviors in comparison to *Linear Feedback Shift Registers (LFSR).*
- **Large-scale high-speed asynchronous application**
  Our target is RAPPID, a 120,000-transistor, 3.6 GHz asynchronous version of the Pentium® Pro Instruction Length Decoder [14].
- **No Design-for-Test**
  Although the core design of RAPPID has many levels of sequential logic and is not materially pipelined in nature, no Design-for-Test hooks are included other than some debug features.

RAPPID uses standard static and dynamic synchronous library elements, and adds an aggressive high-speed asynchronous timing methodology, called *Relative Timing* [16]. Testability was considered one of the major risks, in terms of fault coverage as well as area and performance[13, 12]. The testability study started after the design was complete, which explains the absence of Design-for-Test hooks. To minimize the impact on the performance, we opted for a non-invasive approach, outside the RAPPID core, which led us to BIST.

Sections 2–3 give the necessary background on Cellular Automata and RAPPID. The CA-BIST solution follows in Section 4. Costs and fault coverage are addressed in Section 5, and we give a short resume of the main achievements in Section 6.

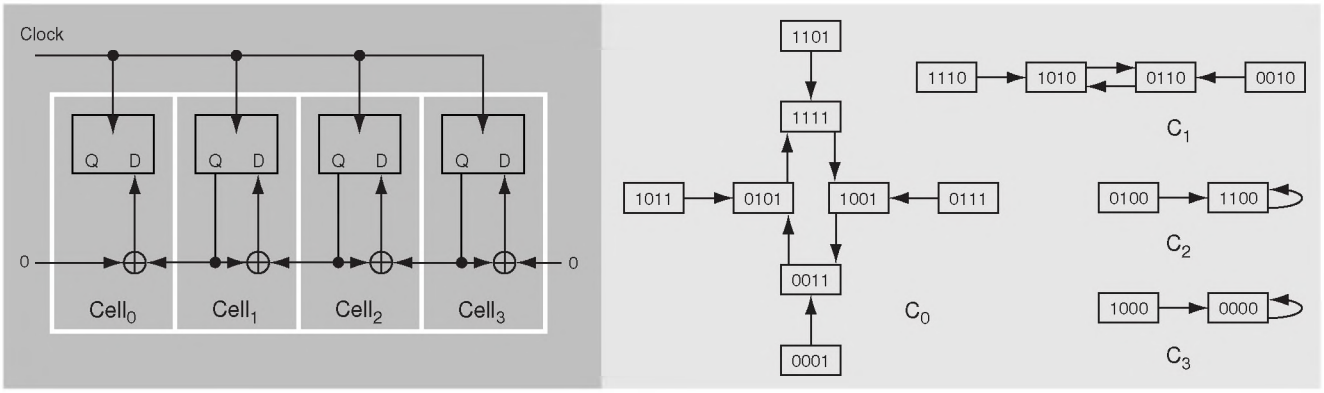


Figure 1 Architecture (left) and state transition diagram (right) of $D1^*CA$ with rule vector $<90, 102, 102, 102>$ (see Table 1). Boundary bit values for left and rightmost cells are assumed 0. This circuit implementation uses globally clocked D-flipflops to store the state bits. In the transition diagram, state bits are catenated from left (Cell_0) to right (Cell_3). The state space is distributed over four components: C_0 with a 4-state cycle, C_1 with a 2-state cycle, and C_2 and C_3 with a 1-state cycle each.

Rule	**Next State Table Representation (left self right)**								**Next State Logic Function**
	111	110	101	100	011	010	001	000	
90	0	1	0	1	1	0	1	0	$s[i]_{t+1} = s[i{-}1]_t \oplus s[i{+}1]_t$
102	0	1	1	0	0	1	1	0	$s[i]_{t+1} = s[i]_t \oplus s[i{+}1]_t$
150	1	0	0	1	0	1	1	0	$s[i]_{t+1} = s[i{-}1]_t \oplus s[i]_t \oplus s[i{+}1]_t$
165	1	0	1	0	0	1	0	1	$s[i]_{t+1} = \overline{s[i{-}1]_t \oplus s[i{+}1]_t}$

Table 1 Four examples of 2-state, 3-neighborhood additive CA rules. The state of cell i at transition t is denoted by $s[i]_t$, while $s[i{-}1]$ and $s[i{+}1]$ refer to left and right neighbor states. Symbol $\oplus$ denotes the binary XOR function. Rule numbers relate to the decimal value of their table results, e.g. $90 = 01011010$ (most significant bit left). Note that rules 90 and 165 are complementary.

2 Cellular Automata Test Preliminaries

Cellular Automata, abbreviated *CA*, are regularly interconnected cell arrangements. Each cell acts as a finite state machine with some number of states, and with a next state function that typically depends on the present state of a limited subset of neighborhood cells [18]. In this case study, we focus on simple 2-state, 3-neighborhood CA with next state functions based on left, self, and right cells and implementable with only XOR and XNOR gates and D-flipflops. Theory and applications for these so-called *additive CA* can be found in [6]. This Section outlines some of the background theory for our application: a Built-In Self Test (BIST) solution for the RAPPID asynchronous instruction length decoder, which we present in Section 4.

For a 2-state, 3-neighborhood additive CA there are 8 (2^3) neighborhood combinations but just 7 XOR-based next state functions, also called *rules*, and 7 complemented rules based on XNOR. Table 1 gives four such rules. An example CA with four cells is shown in Figure 1 along with its state transition graph. It belongs to a specific class of CA with rule configuration $< 90, 102, \ldots, 102 >$, referred to as $D1^*CA$. The global transition function of this CA can be represented as a 4×4 *characteristic matrix* T, operating over GF(2) and defined as follows:

$$\begin{aligned} T[i,j] &= 1 \quad \text{if cell } i\text{'s next state depends on cell } j \\ &= 0 \quad \text{otherwise} \end{aligned}$$

If state s_t at the t^{th} transition of this 4-cell $D1^*CA$ is represented as column vector, then the characteristic matrix and next state s_{t+1} are given by:

$$\begin{aligned} T &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ s_{t+1} &= T \cdot s_t \end{aligned} \qquad (1)$$

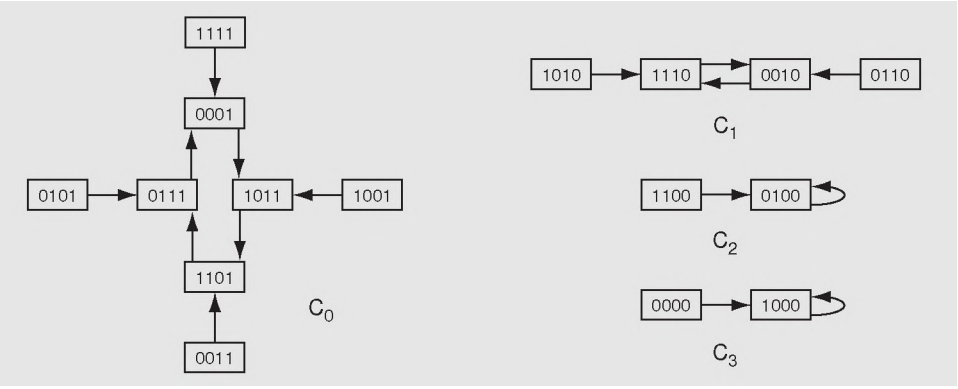


Figure 2 State transition diagram for $D1^*CA_{dual}$ with rule vector $< 165, 102, 102, 102 >$, the dual version for $D1^*CA$ of Figure 1. Note that the state space partitioning into four components is similar, but that the cyclic states in the dual version are non-cyclic for the normal version, and vice versa. For instance, the cyclic state 1011 in component C_0 for $D1^*CA_{dual}$ is outside the cycle for $D1^*CA$, whereas the non-cyclic dual state 1111 is again cyclic for $D1^*CA$.

CA belonging to the class $D1^*CA$ have some unique properties that can be exploited for on-chip test pattern generation and for synthesizing self-testable sequential circuits [6, Chapter 9]. In our case study we focus on test pattern generation. The capability for generating low-cost test suites with high fault coverage stems from the relation between $D1^*CA$ and its dual version $D1^*CA_{dual}$, whose transitions complement one another and can be combined cost-effectively. Figure 2 shows the dual version for the four-cell $D1^*CA$ introduced earlier. It uses an XNOR instead of the XOR in the rule for the leftmost cell, resulting in the rule configuration $< 165, 102, \ldots, 102 >$. The next state GF(2) function for $D1^*CA_{dual}$ is given by:

$$s_{t+1_{dual}} = T \cdot s_t + \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad (2)$$

Note that the transition diagrams for $D1^*CA$ and $D1^*CA_{dual}$ have similar state graphs, consisting of four components with the same set of states per component. However, cyclic and non-cyclic states are interchanged between the two versions. As such, the two CA will traverse different states in a component. Interpreting the state as a test pattern, they will thus generate different sequences of test patterns. If in addition to traversing state transition graph components in one CA it would be easy to switch back and forth between the two CA, then a wide variety of test patterns could be realized economically with a simple CA combination on chip. In Section 4.1 we present our on-chip test pattern generator which is built around an 11-cell $D1^*CA$ and its dual version, and we show that switching back and forth is both cost-effective from an algorithmic perspective as well as in terms of circuitry.

3 Asynchronous Instruction Length Decoder

This CA-BIST case study is part of the RAPPID project, investigating aggressive high-performance asynchronous circuit design [14]. A prototype iA32 instruction length decoding and steering unit was designed and fabricated on a 0.25 micron CMOS process. The circuit was targeted for performance, with measured throughput of 2.5–4.5 instructions per nanosecond. An aggressive circuit design style was developed in the project that added timing to the asynchronous protocols [16]. Domino gates were used as latches for much of the state in the control and data paths. This permitted us to increase the throughput and decrease the area of the design, at the expense of additional protocol analysis to validate the timing constraints.

Building a test chip forces one to implement a complete design. This helped us to better understand the design methodology and test the maturability of these ideas [17]. Testability was considered a major risk to the design in terms of ability to achieve reasonable coverage, area penalty, and performance penalty. Any logic placed in the evaluation path will lie on a critical path of the asynchronous circuit due to its reactive nature. Significant Design-for-Test effort would be required to try and move test logic into the reset or return-to-zero handshake logic that is out of the critical path. The CA-BIST approach was designed to have minimal impact primarily on the performance of the circuit, but also on the area.

Section 3.1 below explains how the overall architectural structure of RAPPID has been scaled to high performance, and where the CA-BIST interface lies. Section 3.2 gives some background on the asynchronous protocols and domino gates, used in the design and revisited in Section 5.1 when we discuss the fault coverage results.

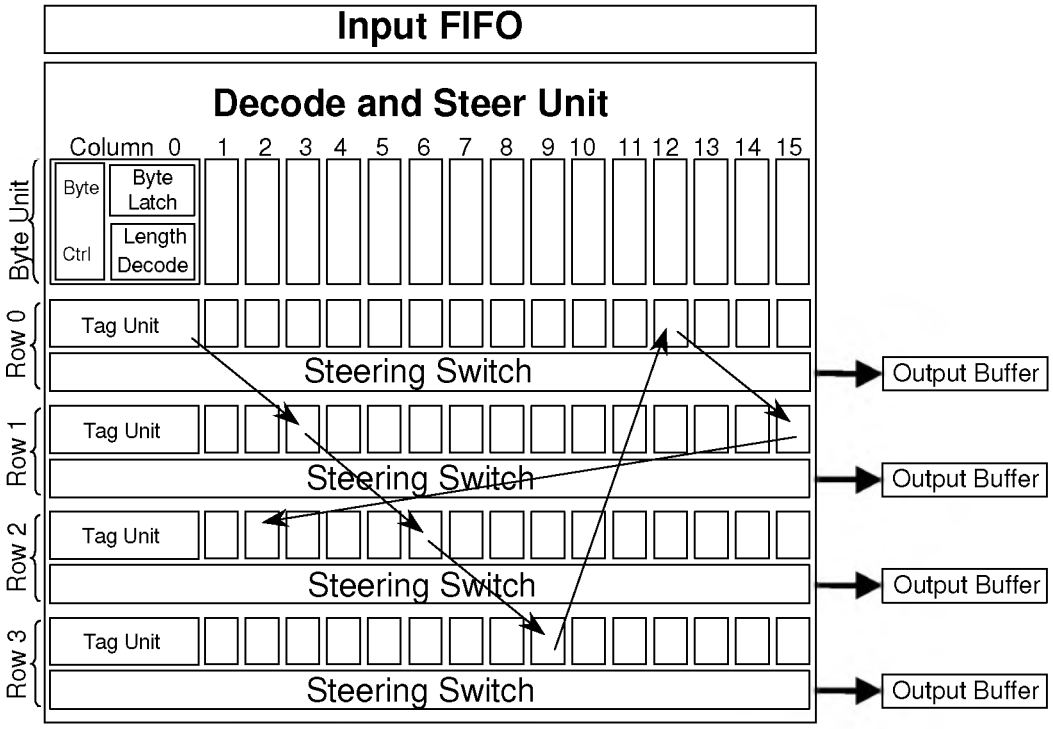


Figure 3 (RAPPID Architecture) The Input FIFO holds a 16-byte wide instruction cache line, containing 5 instructions on average. Instructions are decoded in parallel over 16 byte Decode Columns with an average rate of 0.72 GOPS, giving a total average of $5 \times 0.72 = 3.6$ GIPS. To maintain this average at the output side, a 4-step top-down Tag cycle gathers and distributes the instructions over 4 Output Buffers, each operating at 0.9 GIPS. The sequence of arrows in the picture illustrates the 3.6 GIPS instruction flow through the Tag Units for a typical scenario with 5 length-3 instructions.

3.1 RAPPID Architecture

The RAPPID test chip implements instruction length decoding for the Pentium®Processor instruction set [9]. Analysis showed that instructions longer than seven bytes are rare, and that certain instructions appear with much higher frequency in common programs. Our asynchronous design optimized for these common cases at the expense of unoptimizing rare instructions. Instructions longer than seven bytes and length modifying prefixes, for instance, may have a four-fold or greater reduction in performance compared to a common instruction.

RAPPID receives a 16-byte wide instruction cache line at its input, extracts the instructions, and places each instruction separately in the output buffers. As shown in Figure 3, sixteen parallel length decoders are employed, which speculatively compute the length in each column as if a new instruction began at each byte position. A torus-like distributed tagging and crossbar switching circuit with 16 columns and 4 rows packs the bytes into instructions and steers them into four output buffers. These row and column dimensions are designed to balance the average computation rates of the three operations: 720 MHz for length decoding, 3.6 GHz for tagging, and 900 MHz for steering. The scaling of this architecture by replicating circuitry in one or two dimensions increases the parallelism in the design much like pipelining, though unlike standard pipeline concurrency it does not increase the latency.

The testability study of RAPPID started after the design was complete. Other than the Input FIFO, which is used only as an interface to the tester, and some debug features to freeze and inspect a limited set of internal state signals, no Design-for-Test features had been applied. To avoid performance penalties or re-designs, our options were restricted to non-invasive test structures. We chose for a CA-BIST approach external to the RAPPID circuitry. The CA Test Pattern Generator is integrated in the Input FIFO, while the CA Response Evaluator observes the Output Buffers. The BIST logic minimally impacts latency and throughput for the instruction length decode operation. No test circuitry is included in the core of the design other than some routing and drivers to make the Output Buffers observable for signature analysis by the Response Evaluator. Further details on the CA-BIST architecture and algorithm can be found in Section 4.

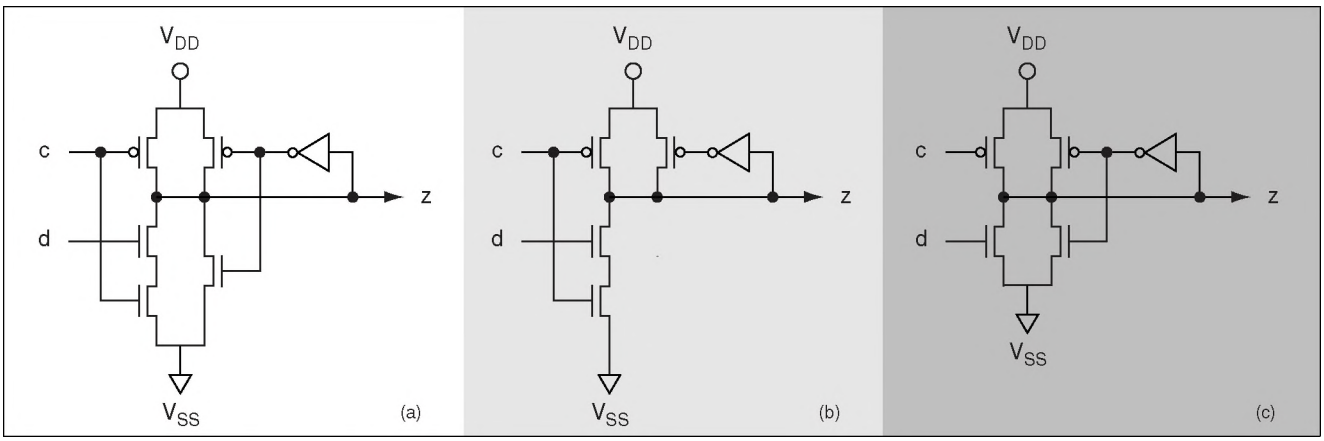


Figure 4 Three types of domino gates used in RAPPID, and sourcing most of the CA-BIST fault coverage escapees. All three examples implement a set-reset latch with data evaluation signal d for 'set' and control signal c for 'reset' (domino reset states are typically high). The weak inverter feedback loop, or *keeper*, sustains the output state z when neither input signal is driving. The *full keeper* in versions (a) and (c) sustains both high and low states, whereas the *half keeper* in (b) only sustains the high reset state. We typically use implementation (b) for conservative operations where the data remains set until the gate resets. Set and reset can overlap for *footed* versions (a) and (b), but have to be mutually exclusive in *unfooted* version (c) where the c-controlled nMOS transistor (*foot*) is lacking. Version (c) is typically used for pulse domino operations (see Figure 5).

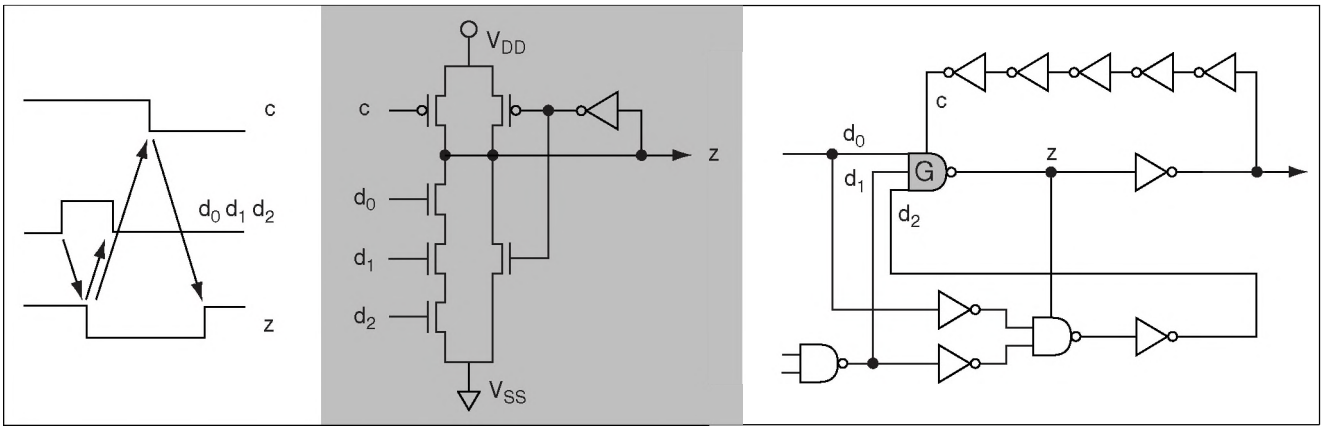


Figure 5 Example of pulse domino logic as found in the Length Decoder of RAPPID. The implementation for NAND gate G in the right-hand schematics is given in the grey-colored area in the middle, and a typical set-reset cycle is shown left. As soon as all three data signals are high, gate output z is set low. The down-transition on z triggers two self-resetting loops, which in parallel end the 'set' phase and start the 'reset' phase. The resulting three gate delay 'set' pulse $d_0 d_1 d_2$ is sufficiently narrow to avoid overlap with the seven gate delayed self-reset for c, and wide enough for z to set subsequent domino stages.

3.2 Asynchronous Circuit Details

Synchronization and communication in RAPPID goes via handshaking. Besides the standard 4-phase request-acknowledge protocols we use bundled data protocols. An aggressive design style, called *Relative Timing*, is used to customize these protocols [16]. For instance, we allow pulse signals as handshakes to eliminate the need for complementary handshake circuitry and thus get faster, smaller, and lower power implementations with less test redundancy.

Both control and data paths use a mixture of static and domino logic, with the domino gates operating as set-reset functions for synchronization and communication. Figure 4 shows three types of domino gates that contributed most to the CA-BIST fault coverage escapees, and which will be further analyzed in that respect in Section 5.1. A pulse domino example can be found in Figure 5. We would like to stress that the circuits in Figure 4 are not new. Versions (a) and (c) are well-known asynchronous gates [10]. All three combinations are commonly used in domino designs [8].

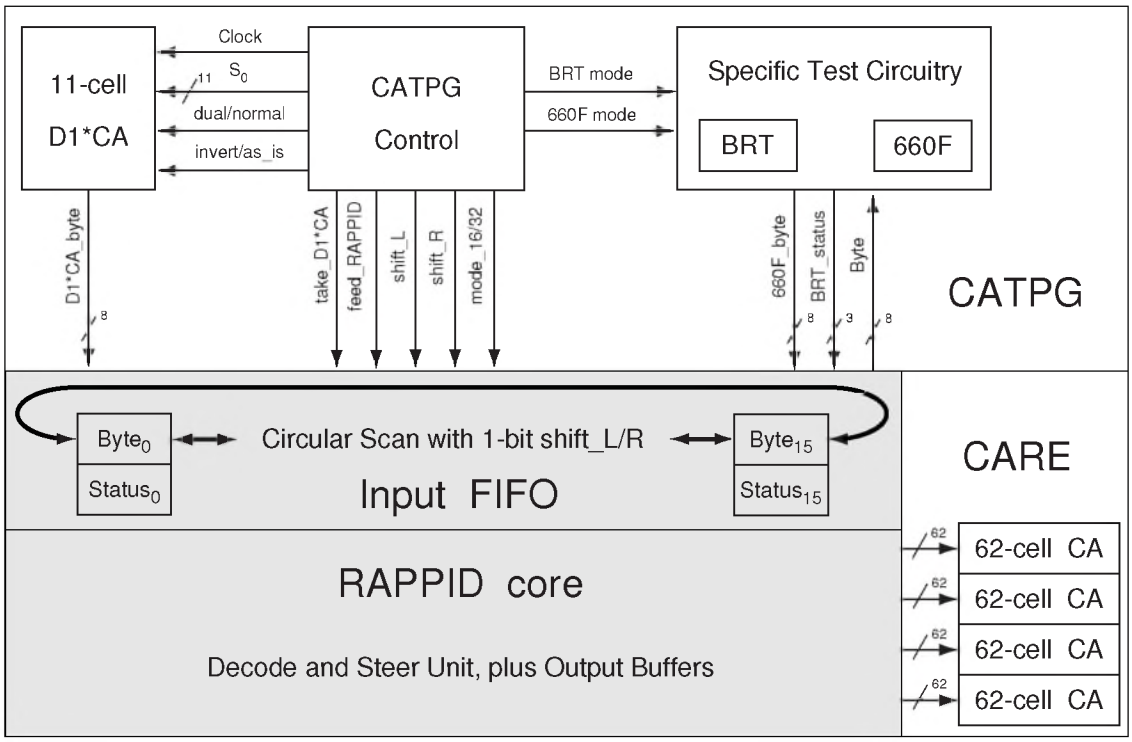


Figure 6 (CA-BIST Solution) The grey area matches the original RAPPID design before BIST came into the picture.

4 CA-BIST Solution for RAPPID

The CA-BIST circuit lies outside the RAPPID core, primarily to minimize its impact on the design performance. Figure 6 gives an architectural overview of the CA-BIST solution, showing the two main modules:

- **CA-based Test Pattern Generator (CATPG)**
  We use a bootstrap process to generate the test patterns. The process starts from an 11-cell $D1^*CA$ and its dual version. As already indicated in Section 2, such a CA pair can generate a wide variety of test sequences. For our purpose, the CA-generated test patterns are chopped into 16-byte wide cache lines for the Input FIFO. Each pattern is used 256 times, every next time shifted over one more bit position: shifting one round-robin left and another round-robin right. A circular scan chain (already present in the Input FIFO) with both left and right rotation supports this.

- **CA-based Response Evaluator (CARE)**
  We generate a signature for each 62-bit Output Buffer (7 instruction bytes plus 6 additional bits with length, prefix and long instruction information). All 4 signature analyzers are implemented as 62-cell maximum-length CA based on rules 90 and 150 (see Table 1) and have an aliasing probability of 0.5^{62}. Analysis techniques based on this type of CA are discussed in [15, 7, 6] where also an isomorphism with Linear Feedback Shift Register (LFSR) solutions is proven. We upgrade the signature for every CA-generated test, which is once for every 256 test patterns executed on the RAPPID core (see CATPG discussion above).

Both modules allow testing at speed to create realistic operational conditions inside the core, which is particularly crucial for testing the pulse domino operations. The following Section explains the CATPG architecture and algorithm in more detail. Contrary to the CA that we use in CARE, the 11-bit $D1^*CA$ in the CATPG module cannot be implemented as LFSR because it has non-cyclic state transition behavior, as we will see. There will be no further elaboration on the implementation details of the CARE module, which are standard and can be found in the given references.

4.1 CATPG Architecture and Algorithm

Above, we sketched the bootstrap process in the on-chip CA-based Test Pattern Generator (CATPG). Details are explained below, starting from the test sequences generated by the 11-cell $D1^*CA$ and its dual version. Because the Input FIFO acts as interface between the CATPG and the RAPPID core, we will first give a quick update on the input instruction flow through the FIFO and on the corresponding CATPG operations.

- **Circular Scan**
  Data in the FIFO can be recirculated to supply a continuous stream of cache lines to the RAPPID core for performance and power measurements. Loading and recirculation are accommodated by connecting all data bits in a circular scan chain, which is also used for the CATPG bootstrap operation.

- **Branch Instructions**
  In addition to the 16 instruction data bytes, the FIFO contains 3 additional bits for each byte, indicating the status of the byte for branch prediction: *unused, branch, or target*. Normally, these status bits are derived from a separate Branch Target Buffer. If a cache line contains a branch instruction, the first byte of the branch instruction is tagged as 'branch'. All bytes after the branch instruction up to the branch target in the next cache line get status 'unused'. And the branch target instruction is tagged as 'target'.

  To exercise branch instructions, the CATPG has direct access to the status bits which are defaulted to non-branch indicators. In test mode, the CATPG interprets data patterns of the form $0111xxxx$ as 2-byte branch instructions with opcode $7x$ (hexadecimal), and with the target byte at the same position in the next cache line. There is a simple test circuit, called *BRT* for 'BRanch Test', which is only active in test mode. If a CA-generated cache line contains the above branch instruction, the BRT tags the leftmost occurrence of the branch as 'branch'. It marks all 3-bit status registers after this branch 'unused' for the next 16 bytes, and tags the following byte as 'target'.

- **Modes of Operation**
  The Instruction Length Decoder supports 16-bit instruction mode as well as 32-bit instruction mode. The default instruction mode is set globally for the entire architecture. In addition, the mode can be changed locally from 16 to 32 bit and vice versa within the instruction stream, via a so-called *length modifying prefix* (opcodes 66–67). Both these modes of operation and their settings need to be tested. Assuming that the local mode setting will be covered by the CA-generated test patterns, it suffices to give the CATPG access to the global mode setting.

Section 4.1.1 presents the implementation details of the 11-cell $D1^*CA$ and explains how the initial test sequences for the bootstrap process are generated from this CA module. The following Section 4.1.2 shows how the other CATPG modules in Figure 6 join $D1^*CA$ in the test generation process. Here, we also explain where our on-chip test generation approach fell short, and how we improved upon it.

4.1.1 ONE for ALL: 11-cell D1*CA

Our CATPG solution is tuned to the 16 replicated columns in the Length Decoder (see Figure 3). Replication in the design is mimicked in the test generation process by circulating the test patterns from column to column. Test patterns are generated from $D1^*CA$ and its dual $D1^*CA_{dual}$, as we mentioned in Section 2. We chose the smallest $D1^*CA$ with at least 8-bit wide states and cycles of length 16, so that we can fill 16 columns with different byte patterns. Figure 7 shows the two Components of this 11-cell $D1^*CA$ that we used as test generation engine for RAPPID. Implicit requirement is that the 32 fillings generated by this pair are generated on chip, which requires additional control to

- set the start state, and
- switch between the normal and dual Component.

A similar problem is faced in CA-based approaches for synthesizing self-testable sequential circuits. Our solution, *although new*, was inspired by those in [6, Section 9.3]. The new algorithm for traversing all 32 cache fillings that this Component pair can generate is as follows.

Traversal Algorithm

Step 1
Select a cyclic state S_0 in the normal Component.

Step 2
Invert least significant bit (Cell_{10}), resulting in S_1.

Step 3
Switch to the dual Component, run a full cycle from and to S_1, and use the state sequences as test inputs.

Step 4
Switch back to the normal Component, make a single transition, and let S_2 be the resulting state.

Step 5
Run a full cycle in the normal Component, from and to S_2, and use the state sequences as test inputs.

Step 6
If S_2 is different from the initial state S_0 in *Step 1* then continue with *Step 2*, else stop.

It can be proven that the above algorithm terminates after exactly 16 loops of *Step 2–6*, and that every loop generates two new 16-state wide test sequences. Thus, all 32 different fillings are generated. Figure 8 shows the additional circuitry inside the on-chip 11-cell $D1^*CA$ to accommodate this algorithm. The choice of S_0 and also the global control for bit inversion in *Step 2* and Component switching in *Steps 3–4* and for clocking the transition cycles are handled by the CATPG Control module (see Figure 6). Note that the full cycles in *Steps 3,5* circulate replica test sequences (shifted byte-wise) along the columns of the Length Decoder. This replication is taken one step further in the next Section, where a bootstrap process circulates bit-shifted versions of these test sequences along the columns.

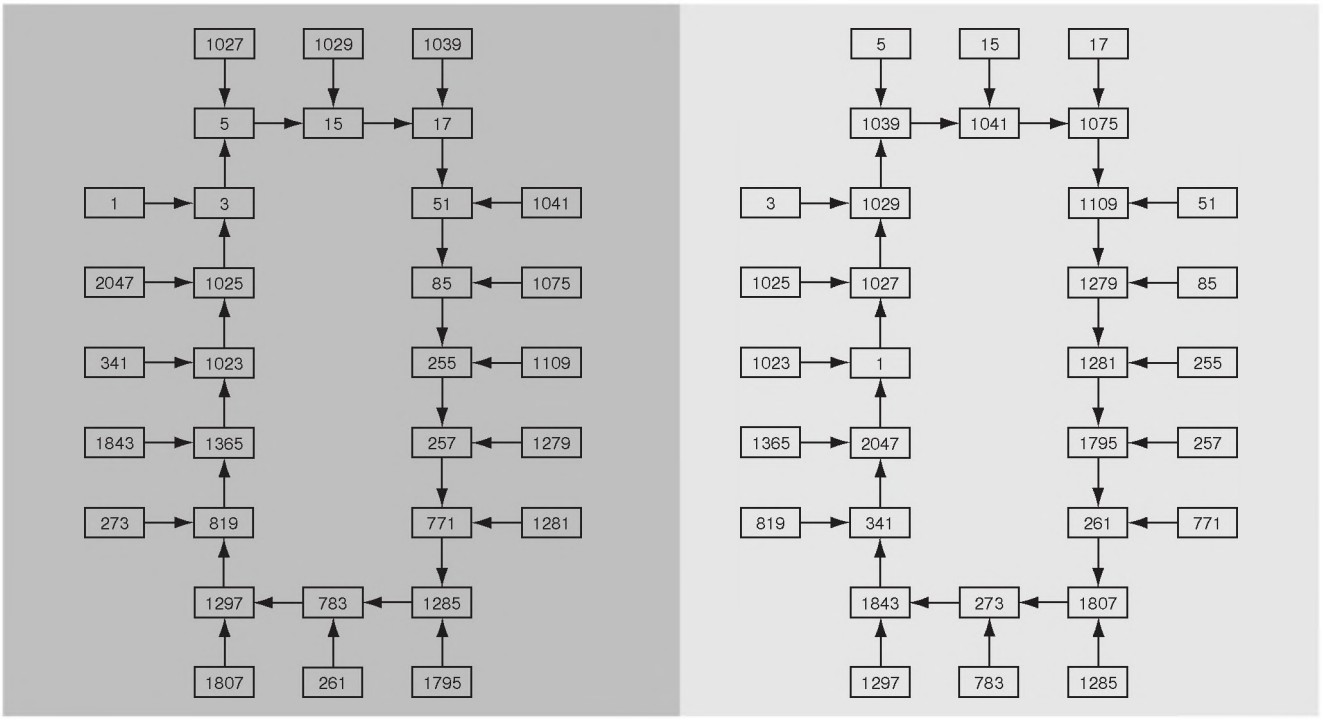
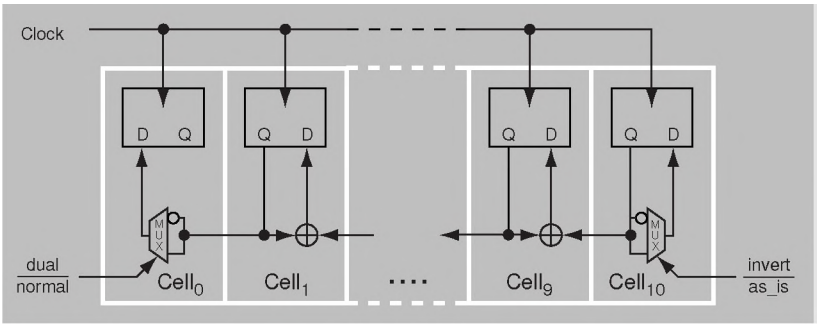


Figure 7 These Components of an 11-cell $D1^*CA$ (left) and its dual (right) form the engine of RAPPID's bootstrapped test generation process. State values are shown as decimals as opposed to Figure 1, e.g. $5 = 00000000101$ with the most significant bit for Cell_0 at the left and the least significant bit for Cell_{10} at the right. Both CA cycles have length 16, and there are 16 different states to start from. Each cycle is good for filling one cache line in the Input FIFO with "instruction" bytes (only the least significant 8 bits of a state are passed on). Thus, a total of 32 different cache lines (16 per CA) is generated.



	Cycle Length				
	16	8	4	2	1
Components	48	30	3	1	2
States	1536	480	24	4	4

Figure 8 Circuit implementation (left) and state space partitioning (right) for the 11-cell $D1^*CA$ plus dual. The circuit has two additional multiplexers and mode signals for running the Traversal Algorithm of Section 4.1.1. The multiplexer in Cell_0 switches between normal mode where $s[0]_{t+1} (= 0 \oplus s[1]_t) = s[1]_t$ and dual mode, for which $s[0]_{t+1} = \overline{s[1]_t}$ (cf. Section 2). The multiplexer in Cell_{10} provides the bit inversion for *Step 2* in the algorithm. The circuit generates 2048 different states, partitioned into 84 Component pairs of various cycle lengths as indicated at the right. Running the Traversal Algorithm for all Component pairs would fill 2048 cache lines in the Input FIFO. However, we only use the pair in Figure 7 with 32 fillings.

4.1.2 ... and ALL for ONE: CATPG

Owing to the 11-cell $D1^*CA$ and Traversal Algorithm in the previous Section, we have 32 CA-generated cache lines to start the RAPPID testing process. We expand this set on-the-fly into a three orders of magnitude larger set with 32768 tests. The on-chip expansion comprises three steps.

Step 1 *Bootstrap 32 to 8192 tests with Circular Scan*
By design, all 128 data bits in the Input FIFO are linked as a circular scan chain with 1-bit left and right shift options (see introduction to Section 4.1). Using both scan shift facilities, each CA-generated test sequence is bootstrapped into a set of 256 test sequences, as outlined in the following algorithm.

Bootstrap Algorithm

Step 1
Run Traversal Algorithm to fill the Input FIFO with the next 16-byte wide test sequence. Fill by copying each new byte into the leftmost instruction byte, after right-rotating the FIFO 1 byte.

Step 2
Left-rotate the FIFO over 1 bit position, and feed the resulting test sequence to RAPPID. Repeat this $16 \times 8 = 128$ times, coming back to the original CA-generated filling.

Step 3
Repeat *Step 2*, but rotate rightwards this time.

Step 4
If the Traversal Algorithm has not completed then continue with *Step 1*, else stop.

The overall control for shifting leftwards or rightwards as well as synchronization and communication with the $D1^*CA$ Traversal Algorithm are handled by the CATPG Control module (see Figure 6).

Step 2 *Close test gaps with Specific Test Circuitry*
At the beginning of Section 4.1 we introduced a simple test circuit, called BRT for 'BRanch Test', which makes it possible to test the branch instruction logic inside RAPPID. The need for a BRT circuit was known in advance, which is different for the following case that we discovered during development of the BIST. A large gap showed up in the coverage of instruction types when running the 8192-sequence test suite generated in **Step 1** in the three modes available at this point: 16-bit instruction mode, 32-bit instruction mode, and branch mode (with BRT active). The gap related to 2-byte opcodes of long length instructions, which start with a length-modifying prefix (e.g. *66*) followed by a 2-byte opcode (e.g. *0F*). We created a specific test circuit *660F* to force and circulate the combination *66-0F* on pairs of subsequent columns (one pair per test sequence), which runs under a separate mode and closes the gap.

Step 3 *From 8192 to 32768 tests in 4 Operation Modes*
The CATPG module runs the Bootstrap Algorithm in **Step 1** four times, each time in a different mode:

- *16-bit instruction mode*
  One of the global functional modes in RAPPID, accessible for CATPG (see begin Section 4.1).
- *32-bit instruction mode*
  Another global functional mode for RAPPID, that can be accessed by the CATPG module.
- *branch mode*
  In this mode, the specific test circuit BRT becomes active, and upgrades the test sequences with branch information.
- *660F mode*
  As explained in **Step 2**, this mode was added to fill a large gap in the RAPPID test instruction flow. Similar to the branch mode, it upgrades the bootstrapped test sequences before they are shipped to the RAPPID core.

To get a fast first-order qualification of the test suite, we measured the percentage of covered min-terms in the Length Decode PLA of the instruction set. The PLA reflects the control flow in the abstract hardware description model for the instruction length decoder. In this model an instruction stream enters the decoder and is broken into independent streams of instructions, which are steered to the Output Buffers. The PLA min-terms encode the way the instruction bit fields are examined and broken up. Taking these min-terms as a coverage metric gives a well-defined target that is independent of implementation details. In this way, we discovered the test gap addressed in **Step 2** above, for which we added the test circuit *660F* that covers the escapees. All PLA min-terms are now covered by the CATPG algorithm. However, as one would expect, we missed some of the particular implementation details that this metric did not target. These are discussed in Section 5.1.

5 Cost and Performance

We evaluate the costs and effectiveness of our CA-BIST solution by looking at the area and speed impact on the design, and by fault-grading the test suite on the actual switch-level schematics. Due to schedule constraints, the fabricated version of RAPPID does not include CA-BIST. However, the full design was implemented — with the BIST partly in layout, partly as gate and transistor level schematics — and simulated to estimate the impact on timing and area. Interfacing with RAPPID occurs at the natural design boundary, and thus no logic or Design-for-Test modifications were applied to the RAPPID core (see Figure 6). Scan facilities for CATPG and debug are shared, and integrated in the FIFO as Circular Scan Chain.

The combination of CA-BIST and debug logic induces a small penalty, estimated at 5% for area. The CARE signature analyzers can uninvasively observe the data values on the output buffers. This extra load is very small and does not significantly change the delay on the output. Therefore, to the first order, the performance impact is due to the scan input multiplexer on the instruction bytes in the FIFO, resulting in a latency penalty of one gate delay, being 5% (we over-conservatively reported 10% in [14]). Because the CA-BIST is taken off the critical path, throughput is not materially affected.

5.1 Fault Coverage and Evaluation

The RAPPID core contains 120,000 transistors and uses a standard synchronous 0.25 micron CMOS library. Because our test approach targets PLA min-terms being (1) a high-level functional metric and (2) not specifically asynchronous or synchronous in nature, we expect it to miss certain categories of structural faults in the actual switch-level implementation (see end of Section 4.1.2). More specifically, we expect to see uncovered classes related to the asynchronous design style. To fault-grade RAPPID, we used the switch-level fault simulator COSMOS [4], injecting stuck-at faults at each input and output of a CCSN[1]. To keep run times reasonable, we injected the faults in one Length Decoder column only, and only in one Tag Unit in this column. Figure 9 gives an overview of the results. Below, we group these results into detectable faults and faults that cannot be detected with standard fault simulation.

- **Detected and Unexercised faults (86%)**
  CA-BIST covers 81% of the simulated stuck-at faults. Of the remaining faults, 5% are unexercised and relate to missing operands for internal data operations, which are excellent candidates for tailored test pattern generation in addition to BIST. All detectable faults are in these two categories. CA-BIST thus covers 94% of the testable faults.
- **Benign and Suspicious faults (14%)**
  All faults in this group are inside keepers of domino gates, and are basically undetectable in the context of voltage-based unit-delay switch-level fault simulation. About 9% is likely benign and will not decrease the reliability of the domino operation. The other 5% can lead to catastrophic behavior in the presence of cross-talk, coupling and other sources of noise interference. An example of each is given in Figure 10, a benign fault in (a) and a pulse-narrowing fault in (b). Besides pulse narrowing, a significant part of the suspicious faults result in floating gates. For instance, a stuck-at 1 on the domino half keeper in Figure 10(b) will render the keeper powerless, and leave the gate floating during feedback mode.

Our analysis results are very similar to those presented in [1, 5], with the exception that those results are for clocked domino logic, while our circuits are fully self-timed. What is particularly encouraging is that we inherited almost all of the 5% undetected and potentially catastrophic faults from synchronous (pulse) domino logic. Apparently, the added asynchronous timing methodology which asserts RAPPID's high performance and low power does not further impact the testability.

[1]Channel Connected Sub Network (CCSN) is the switch-level equivalent of a logic gate. It contains the set of transistors that source or drain the same net (channel).

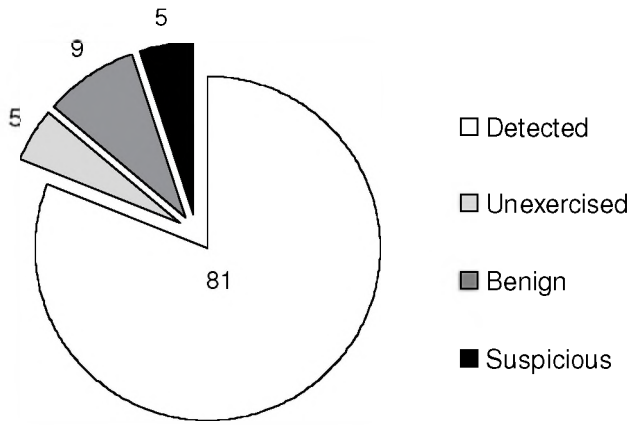


Figure 9 Stuck-at fault coverage distribution for the CATPG test suite. We used the switch-level fault simulator COSMOS to inject faults in one Length Decode Column and one Tag Unit (4000 stuck-at faults in total) and to simulate their fault coverage in the full RAPPID design. Because of design replication (cf. Section 3.1) we expect similar percentages per category for the other rows and columns. In other words, we expect similar figures when injecting all stuck-at faults in the RAPPID core.

6 Conclusion

This CA-BIST study is part of the RAPPID project which explores high-performance asynchronous circuit design for Intel processor architectures. The testability study started after the design was complete, with no Design-for-Test hooks other than some debug features. The non-invasive CA-BIST logic minimally impacts the design latency, throughput, and area, which were our basic requirements. Other than that our purpose was twofold:

- **Achieve high fault coverage with CA-BIST**
  The BIST solution should cover a majority of the stuck-at faults in the design, leaving only a minority for tailored test pattern generation and testing off-chip. This goal has been achieved as our solution covers 94% of all testable faults, and the remaining 6% turn out to be ideal candidates for test pattern generation.
- **Analyse the testability impact of asynchronous**
  The high fault coverage makes further analysis of undetected faults a manageable task. This analysis showed that 14% of the simulated faults are undetectable with standard fault simulation techniques. A majority 9% of these is benign, but 5% may be catastrophic depending on noise conditions. The encouraging news is that the same fault effects can be expected for a similar clocked circuit, without applying Relative Timing [16]. In other words, *testability is NO excuse* to avoid aggressive asynchronous design techniques in addition to high-performance synchronous solutions.

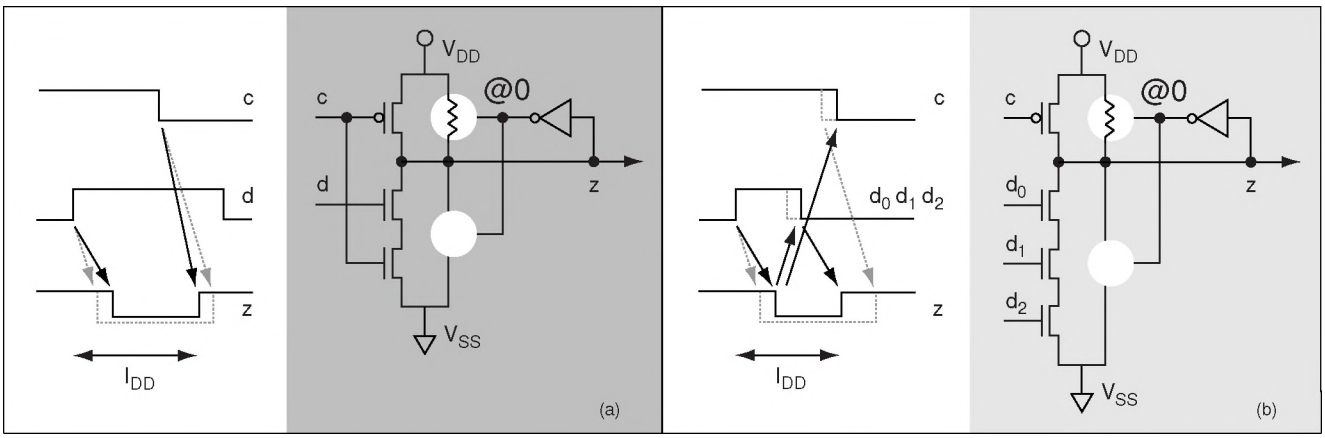


Figure 10 Undetected stuck-at 0 keeper fault which is benign for the circuit in (a) and potentially catastrophic for (b). Both set-reset scenarios show pulse narrowing and slightly deteriorated voltage levels on output z due to elevated I_{DD}. (original behaviors in light-grey, faulty scenario in black; see also Section 3.2). In case (a) there is no need for the keeper to sustain a low 'set' value because d remains valid until the gate is reset by c. The pulse narrowing can typically be ignored for such a wide pulse. It cannot be ignored for situation (b) which relates to the pulse domino operation in Figure 5, and where the pulse width shrinks from 7 gate delays to just 3 gate delays due to the faulty reset when $d_0 d_1 d_2$ is withdrawn. To determine whether the shrink is catastrophic, this scenario should be examined for realistic noise conditions.

Acknowledgements We thank Susmita Sur-Kolay and the students of Bengal Engineering College and Jadavpur University, and also Rob Roy for helping us develop novel CAD tools to support future CA-BIST applications in Intel.

References

[1] R. Adams, E. Cooley, and P. Hansen. Quad DCVS Dynamic Logic Fault Modeling and Testing. In *Proc. International Test Conference*, pages 356–362, 1998.

[2] V. Alves, F. Franca, and E. Granja. A BIST Scheme for Asynchronous Logic. In *Proc. IEEE Asian Test Symposium*, pages 27–32, 1998.

[3] P. Bardell, W. McAnney, and J. Savir. *Built-In Test for VLSI: Pseudo-Random Techniques*. John Wiley & Sons, 1987.

[4] R. Bryant. Boolean Analysis of MOS Circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):634–649, July 1987.

[5] J. Chang and E. McCluskey. Detecting Resistive Shorts for CMOS Domino Circuits. In *Proc. International Test Conference*, pages 890–899, 1998.

[6] P. Chaudhuri, D. Chowdhury, S. Nandi, and S. Chattopadhyay. *Additive Cellular Automata: Theory and Applications - Volume I*. IEEE Computer Society Press, 1997.

[7] A. Das, D. Saha, A. Chowdhury, S. Misra, and P. Chaudhuri. Signature Analyzer Based on Additive Cellular Automata. In *International Symposium on Fault-Tolerant Computing Systems (FTCS)*, pages 265–272, 1990.

[8] W. Hwang, G. Gristede, P. Sanda, S. Wang, and D. Heidel. Implementation of a Self-Resetting CMOS 64-Bit Parallel Adder with Enhanced Testability. *IEEE Journal of Solid-State Circuits*, 34(8):1108–1117, Aug. 1999.

[9] Intel Corporation. *Pentium Processor User's Manual*.

[10] A. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[11] O. Petlin and S. Furber. Built-In Self-Testing of Micropipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 22–29. IEEE Computer Society Press, 1997.

[12] M. Roncken. Defect-Oriented Testability for Asynchronous ICs. *Proceedings of the IEEE*, 87(2):363–375, Feb. 1999.

[13] M. Roncken and E. Bruls. Test Quality of Asynchronous Circuits: A Defect-Oriented Evaluation. In *Proc. International Test Conference*, pages 205–214, 1996.

[14] S. Rotem, K. Stevens, R. Ginosar, P. Beerel, C. Myers, K. Yun, R. Kol, C. Dike, M. Roncken, and B. Agapiev. RAPPID: An Asynchronous Instruction Length Decoder. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 60–70, 1999.

[15] M. Serra, T. Slater, J. Muzio, and D. Miller. Analysis of One Dimensional Cellular Automata and Their Aliasing Probabilities. *IEEE Transactions on Computer-Aided Design*, 9:767–778, 1990.

[16] K. Stevens, R. Ginosar, and S. Rotem. Relative Timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 208–218, 1999.

[17] K. Stevens, S. Rotem, S. Burns, J. Cortadella, R. Ginosar, M. Kishinevsky, and M. Roncken. CAD Directions for High Performance Asynchronous Circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 116–121, 1999.

[18] S. Wolfram. Statistical Mechanics of Cellular Automata. *Rev. Mod. Phys.*, 55:601–644, 1983.